D2

Management program of JAVA application program for embedded system

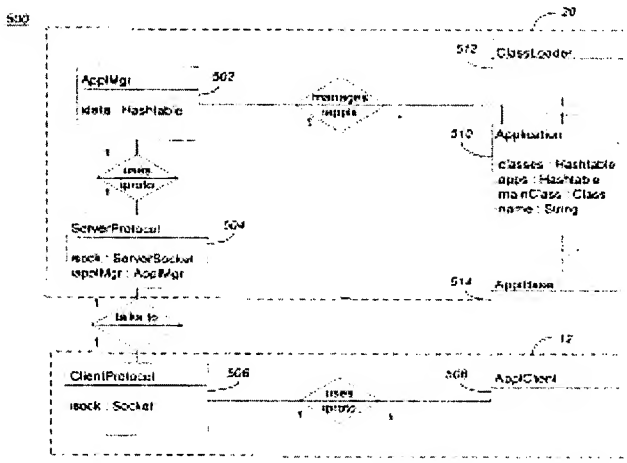
Publication number: CN1265489
Publication date: 2000-09-06
Inventor: JACHE F P (US); DOLAND C-C (US)
Applicant: HEWLETT PACKARD CO (US)
Classification:
- international: ***G06F9/445; G06F9/54;*** **G06F9/445; G06F9/46;** (IPC1-7): G06F9/48
- European: G06F9/445
Application number: CN20001003710 20000301
Priority number(s): US19990259616 19990301

Also published as:
US6430570 (B1)
JP2000250758 (A
CN1197004C (C)

Report a data error he

Abstract not available for CN1265489
Abstract of corresponding document: **US6430570**

An application manager for managing applications in an embedded device is presented. The application manager allows remote control of loading, starting, stopping, unloading, application state querying of applications on an embedded device. Applications are cached in an application cache resident within the embedded device even after application termination to allow for higher efficiency when applications must be unloaded to handle low- or out-of memory conditions.



Management program of JAVA application program for embedded system

Description of corresponding document: **US6430570**

Translate this text

BACKGROUND OF THE INVENTION

[0002] In the embedded device environment, a hardware independent processor such as a Java Virtual Machine is often installed on the device to allow a program to be downloaded and executed on the device. Such a system allows a program written in a hardware independent language such as Sun MicroSystem's Java(R) to be downloaded to any hardware supporting the Java(R) environment in order to customize it for a particular use. This customization is often referred to as the "personality" of the device. Multiple applications may be running in concert within an embedded device to define the device's personality. In this way the device can dynamically be made to function in different unique ways. For example, an embedded device such as a refrigerator may be customized to automatically track its contents. Just as the types of food and naming conventions of similar foods may differ from culture to culture, the embedded device refrigerator may be customized to the particular culture of the users by downloading a new personality describing the food types and naming conventions to the Java enabled embedded refrigerator device.

[0003] In the embedded device domain, a hardware independent processor such as a Java(R) Virtual Machine is installed on the device to allow applications to be downloaded and executed on the device. The Java(R) language provides basic capabilities to build programs to dynamically load new programs. However, the Java(R) language does not provide implementations of program loaders, nor capabilities to manage programs once loaded. The Java(R) language also does not provide a means of downloading and managing applications within a memory constrained embedded device.

[0004] Several existing Java technologies support a program loader type of functionality. Sun Microsystems's Servlet API allows a Java(R) enabled Web server to dynamically extend the capabilities of the server. The Servlet API was mainly created to replace the typical CGI functionality in web servers. The benefits of servlets over CGI type programs include platform independence, reusability (the ability to reuse Java classes through object oriented techniques), performance efficiency (the configurable startup modes of Servlet which allows the same servlet instance to handle many requests (as opposed to the requirement of creating a new process for each call to a CGI script)), and management efficiency (Sun's Java based web server provides a Java admin applet which easily administers the addition of new servlet classes, and the starting and stopping of servlets).

[0005] The Servlet API defines a lifecycle for a servlet by mandating that all servlets implement three methods: init( ), service( ), and destroy( ). The init( ) method is called when the servlet is invoked for first time; the service( ) method is called to handle each request from a client; and the destroy( ) method is called when the servlet is being stopped (i.e. web server shutdown). The Servlet API does not directly support memory management or provide a public package for managing servlets.

[0006] Hewlett Packard's Embedded Java Lab SmallWeb provides a web based interface to Java based objects executing in a JVM. SmallWeb can load objects as needed, and provides a means for objects to export their functionality (through method calls) to web browsers. SmallWeb does not explicitly provide stopping of application objects or memory management features. In addition, SmallWeb usually requires a file system, and for some embedded environments the overhead requirements of SmallWeb may be too large.

[0007] One problem of using Java(R) in many embedded domains is the non-deterministic aspects of the Java(R) memory subsystem. The non-deterministic memory management scheme of the Java(R) language allows the reclamation of un-referenced objects through a garbage collector method gc( ); however, the garbage collector does not specify how or when these objects are reclaimed. Native applications typically force garbage collection to occur by calling the Java(R) Runtime class gc( ) method. If the JVM runs out of memory, an OutOfMemoryError error is thrown. Although not having to manage memory is a benefit of the Java language, most embedded applications need tighter control over how memory is managed than currently provided in the Java(R) language. The tighter control is needed because some embedded applications must continue to execute in low memory situations. This is typically dealt with by implementing a memory manager in the native non-Java embedded applications to ensure that enough memory remains free in order to continue executing.

[0008] Accordingly, a need exists for a universal method for downloading to, and controlling the lifetimes of applications in, a Java enabled embedded device. A need also exists for a memory management handler which handles low- or no-memory situations detected during the execution of applications on the device, and which frees up memory according to a priority-based algorithm.

SUMMARY OF THE INVENTION

[0009] The present invention is a novel system and method for managing the download and lifecycle of an application in a memory constrained embedded device environment. A Java based Application Manager controls the downloading, execution, and caching of Java applications running within a single Java Virtual Machine (JVM) installed on the embedded device. A network capable Application Program Interface (API)

is specified which provides functionality for loading class files, starting, initializing and stopping the execution of applications, and memory management in a Java(R) enabled embedded device. The invention allows embedded devices to be easily re-programmed and managed independently of the type of hardware the embedded device uses.

BRIEF DESCRIPTION OF THE DRAWING

[0010] The invention will be better understood from a reading of the following detailed description taken in conjunction with the drawing in which like reference designators are used to designate like elements, and in which:

[0011] FIG. 1 is a system diagram of a network system in which the invention operates;

[0012] FIG. 2 is a block diagram of an embedded device in which the invention is implemented;

[0013] FIG. 3 is a flow diagram illustrating how an application is brought to life;

[0014] FIG. 4 is an operational flowchart of one embodiment of an application manager implemented in accordance with the invention;

[0015] FIG. 5 is a class diagram of one implementation of an application manager implemented in accordance with the invention;

[0016] FIG. 6 is a flow diagram illustrating the communications between the classes shown in FIG. 5 for performing a load operation;

[0017] FIG. 7 is a flow diagram illustrating the communications between the classes shown in FIG. 5 for performing a start operation;

[0018] FIG. 8 is a flow diagram illustrating the communications between the classes shown in FIG. 5 for performing a stop operation; and

[0019] FIG. 9 is a flow diagram illustrating the communications between the classes shown in FIG. 5 for handling an out-of-memory condition.

DETAILED DESCRIPTION

[0020] A novel Application Manager for an embedded device is described in detail hereinafter. Although the invention is described in the context of a Java(R) environment, it will be appreciated by those skilled in the art that the principles of the invention extend to any system comprising a hardware independent processor for processing code written in a hardware independent language.

[0021] Turning now to FIG. 1, there is illustrated a networked system 10 comprising a computer system 12 in communication with an embedded device 20 over a network 14. Network 14 may be a conventional land-linked wired network such as a LAN or WAN, or may be a wireless network, or any combination thereof.

[0022] Computer system 12 in this system operates as a client, executing a client application 2 that interfaces with server devices (e.g., embedded device 20) over network 14 via a client interface 4. In this example, client application 2 sends requests (e.g., download Appl3, start Appl3, stop Appl2) to embedded device 20, which operates as a server to service the requests. Client application 2 may execute in its own environment on computer system 12, or alternatively within a Java-enabled Web browser (not shown) that contains its own Java Virtual Machine (JVM) (not shown). The web browser interprets Web documents with embedded Java applets that specify the location of the main client application applet class file. The web browser starts up its JVM and passes the location of the client application applet class file to its own class loader 6. Each class file knows the names of any additional class files that it requires. These additional class files may come from the network 14 (i.e., other machines coupled to the network 14) or the client computer system 12.

[0023] Embedded device 20 is a Java enabled device with a Java Virtual Machine (JVM) 22 installed on it. FIG. 2 is a block diagram illustrating the embedded device in more detail. Embedded device 20 comprises a JVM 22, a memory 50 comprising an application cache 52 and a data cache 54, and a network interface 25. JVM 22 comprises a class loader 42 and an execution unit 46. Preferably, JVM 22 also comprises a bytecode verifier 44, although in some memory-constrained devices this may not be implemented. In accordance with the invention, class loader 42 is implemented using the Application Manager 24 of the invention. Application Manager 24 is a Java program responsible for the downloading, execution, and caching of other Java based programs. When JVM 22 begins executing on embedded device 20, it begins executing Application Manager 24.

[0024] Application Manager 24 is a server device operating as a network based class loader able to accept application class files 40 over network 14 via a network protocol 48 and bring the application classes contained in the application class files 40 to life. As known by those skilled in the art, a class file 40 is a file comprising hardware architecture independent bytecodes generated by a Java(R) compiler. The bytecodes are executable only by a JVM, such as JVM 22. Application Manager 24 is a non-typical network based ClassLoader, loading classes 40 from the network 14 on demand. Instead of loading class files 40 through the Java(R) loadClass( ) method as normally done by typical class loaders, the Application Manager 24 instead waits on a server socket or other protocol 48 for a request to download class files 40. When a download request is received, Application Manager 24 receives the class file 40 from the network 14, parses it into classes 28a, 28b, and then calls the Java(R) ClassLoader defineClass( ) method. The class 28a, 28b should be resolved to ensure that it is ready for object instantiation by calling the Java(R) ClassLoader resolveClass( ) method.

[0025] FIG. 3 illustrates how an application 26a, 26b, 26c is brought to life. As illustrated, this is accomplished by instantiating an application's Java class 28a, 28b, creating an instance object 30a, 30b, 30c of the class 28a, 28b, and then calling a main method on the object 30a, 30b, 30c to run the applicatior 26a, 26b, 26c. Multiple instances of the same application may be executed simultaneously by creating multiple instances of the same Java(R) class. For example, as shown in FIG. 3, objects 30a and 30b are instances of the same class 28a, whereas object 30c is an instance of a different class 28b. Accordingly, applications 26a and 26b are different instances of the same application which may run simultaneously, while application 26c is an entirely different application.
[0026] Application Manager 24 provides downloading, starting, stopping, querying, and memory management capabilities. In order to provide these capabilities over network 14, embedded device 20 includes a network interface 25 implementing a network protocol 48 which allows clients 12 to send requests to the Application Manager 24 of the embedded device 20. In the preferred embodiment, network interface 25 speaks to Application Manager 24 using a universal Java language Application Program Interface (API) specified by Application Manager 24. The universal API provides remote devices such as computer system 12 with the ability to specify the desired management of the device 20. Appendix A lists the APIs defined in the illustrative embodiment of the invention.
[0027] In particular, Application Manager 24 allows new applications to be downloaded to the device 20 over network 14. In the preferred embodiment, this is achieved via the API's loadApplClass( ) method. The loadApplClass( ) method handles the downloading and verification of Java class files 40, the loading of the application classes 28a, 28b contained in the class files 40 into the Java(R) environment on the device 20, and prepares the classes 28a, 28b to be executed by instantiating application objects 30a, 30b, 30c using either a Java interpreter or a Just in Time (JIT) compiler (not shown). Application Manager 24 also manages the starting and stopping of Java applications 26a, 26b, 26c on embedded device 20 to allow a user on computer system 12 or other networked device to control the execution of Java applications 26a, 26b, 26c loaded on the embedded device 20. In the preferred embodiment, this is achieved via the startApplo method and stopAppl( ) method of the universal API. Application Manager 24 also allows the querying of application information such as which class objects 28a, 28b are currently loaded, which classes have application instances 30a, 30b, 30c, and what execution state each application instance is in (e.g., "initialized", "executing", "terminated"). This is achieved in the preferred embodiment via methods applClasses( ), applications( ), and applInstances( ) respectively. The names of each loaded class object 28a, 28b is stored in an application list 23 as an attribute, preferably as a hash table, within the Application Manager 24 object. Application Manager 24 also allows querying of information from the Java environment such as the amount of free memory and the amount of total memory. In the preferred embodiment, this information is queried using the freeMemory( ) method and totalMemory( ) method. Application Manager 24 also provides memory management capabilities, including the ability to set limits on the required amount of free memory before a loaded application must be unloaded and the ability to set the order in which to unload cached applications. This is achieved in the preferred embodiment using the setFreeMemoryLimit( ) and setFreeAppsFirst( ) methods. The free memory limit and order in which to unload applications may be queried using the getFreeMemoryLimit( ) and getFreeAppsFirstPolicy( ) methods. This information is storec in a hash table within the Application Manager 24 object, either as a separate field for each application list 23 entry, or as a separate unload priority list 21.
[0028] Network protocol 48 is used to provide network access to Application Manager 24 via network interface 25. Network protocol 48 may be implemented according to any one of several alternative embodiments.
[0029] In a first embodiment, network protocol 48 is implemented using Java(R) Remote Method Invocation (RMI). Java RMI allows direct method calls to be made on objects residing in other JVMs. RMI typically requires a significant amount of memory to support object serialization, identification, and method dispatch. If the embedded device supports the high memory overhead of running RMI, including supporting the running of the RMI name server, RMI is a good implementation choice if the client application interacting with the embedded device 20 and Application Manager 24 is Java based. Bridges do exist between other distributed object mechanisms (e.g., CORBA, DCOM) and RMI, allowing clients to be written in other languages as well. Using RMI, the Application Manager 24 can accept direct calls from remote clients 12 to load, start, and manage applications 26a, 26b, 26c. A simple example of an RMI based Application Manager interface is provided in Appendix B.
[0030] In a second alternative embodiment, network protocol 48 is implemented using network sockets (e.g., TCP/IP socket protocol). If Java(R) is used on the client 12 side as well, it is beneficial to wrap the protocol 48 within a Java class (or classes) to ensure that the protocol 48 remains consistent on both the client 12 side and the server (embedded device 20) side. In order to ensure that the network protocol 48 does not perform blocking reads of the network causing either the client 12 or Application Manager 24 to possibly hang, a wait timeout is preferably set for read( ) operations via the Java(R) Socket.setSoTimeout(method or by using the Java(R) InputStream.available( ) method to determine how much data may be read without blocking.
[0031] In one implementation of such a network socket protocol scheme a single byte which identifies the method to be called, followed by additional data decoded by a wrapper for that specific method, is transmitted. For example, to download a class file 40, the client 12 transmits a 'D' character byte to signify

that the request is for a download, followed by the total length of data being transmitted, followed by the network address of the embedded device 20. In another implementation, the actual data is a null-terminated character class name, followed by a byte signifying whether the class contains a main method or not, followed by bytes containing class data. An object model, supporting a virtual decode method is created to perform the protocol decoding. Returning method call results to the client 12 may also be encapsulated, with a similar protocol being used. In this way results are returned asynchronously from requests, and the client 12 can actually operate as a network server for results. If the client 12 is only talking to one embedded JVM 22, and is idle waiting for results, a simpler scheme can be used where the client 12 waits (with a timeout) for results to come back synchronously.

[0032] In the preferred embodiment, Application Manager24 is implemented as a Java network server application, which accepts requests coded into an application protocol 48 such as Java(R) Remote Method Invocation (RMI) protocol, HyperText Transport Protocol (HTTP) for Web hosted requests, or an applicatior level protocol over TCP/IP sockets. As previously described, one type of request in the protocol 48 is the downloading of an application's Java class file 40. In the preferred embodiment, downloading of class files 40 are performed by calling method loadApplClass( ) or method loadAndInit( ). As previously described, a class file 40 is a binary Java bytecode stream used by the JVM 22 to create a Java class object 30a, 30b, 30c. In addition to downloading the application class file 40 into memory 50, any base classes that the application 26a, 26b, 26c inherits or that interface with the application classes 28a, 28b contained in the class file 40 are also downloaded into an application cache 52 in memory 50. Because the Application Manager 24 allows for caching of class files 40 in memory 50, the application protocol 48 allows for the execution of an already downloaded class file residing in memory 50 via method initAppl( ) followed by a startAppl( ) method call. Once a class file 40 is downloaded, the Java(R) ClassLoader API is used to instantiate the class objects 28a, 28b associated with the class file(s) 40. The ClassLoader.defineClass( ) method constructs a class object 28a, 28b from the class file byte array. Once defined, the class must be resolved to perform class linking, which allows instance object creation and methods calling. The resolution process is initiated by calling the Java(R) ClassLoader.resolveClass( ) API.

[0033] Once a class 28a, 28b is loaded and defined, an instance object 30a, 30b, 30c of the class 28a, 28b is created, in the illustrative embodiment through the Java(R) Class.newInstanceo API. The object 30a, 30b, 30c is needed to call the desired method to start the application 26a, 26b, 26c. In order to call newInstanceo, the class 28a, 28b must have a default constructor which takes no arguments.

[0034] The application 26a, 26b, 26c is executed by calling the desired method on the previously created object 30a, 30b, 30c, passing any arguments to the method. Application Manager 24 must know the method to call on the object and what arguments (if any) to pass. This may be accomplished in one of several ways.

[0035] In one embodiment, all applications are required to implement a Java interface which specifies the method to call. A sample interface is shown below:

[0036] public interface ApplBase

[0037] {

[0038] public void main(InetAddress o);

[0039] }

[0040] This interface specifies an instance method, "main( )", which is the first method called when an application 26a, 26b, 26c executes. Because it is an instance method, an object 30a, 30b, 30c of the class 28a, 28b must be created by Application Manager 24 prior to executing maino. This object may be a Java (R) ApplBase object in the Application Manager 24 because only main( ) is being called. In this example, the main( ) method receives an InetAddress argument, which contains the network address of the client 12 which downloaded the class. In this way, the application can communicate back to the client 12 through a known port if needed.

[0041] In another embodiment, if the JVM 22 supports reflection, for example through Java(R) RMI, the method name is downloaded to Application Manager 24 and reflection is used to search the application class for the desired method. Once the class 28a, 28b is loaded, the Java(R) getDeclaredMethod( ) is called on the class to find the desired method. GetDeclaredMethod( ) returns a Method object, which can ir turn be used to execute the method by calling invoke( ).

[0042] In yet another embodiment, a method which extends from a common base class that all applications inherit from is called.

[0043] Once executing, Application Manager 24 records the new running state of the application 26a, 26b, 26c. Each application class 28a, 28b includes a respective instance list 29a, 29b which identifies each instance that exists of its class. Each entry in the instance list 29a, 29b includes an execution state variable that is updated by Application Manager 24 during the lifecycle of the instant application 26a, 26b, 26c.

[0044] Application Manager 24 handles all Java(R) exceptions or errors occurring in the application 26a, 26b, 26c which would otherwise cause the JVM 22 to terminate. Upon application termination, Application Manager 24 records that the application 26a, 26b, 26c is no longer running (by updating the execution state corresponding to the instance identified in the instant list 29a, 29b), and causes the class object 28a, 28b to be garbage collected if the class 28a, 28b is not to be cached in memory 50.

[0045] Caching of classes 28a, 28b is a unique feature of the invention that allows for better performance. Instead of downloading the application class each and every time an application 26a, 26b, 26c is to be

executed, the Application Manager 24 preferably caches the class object 28a, 28b by default and unloads the class 28a, 28b only when explicitly requested using the unloadAppl( ) method or when the memory management handler 27 selects the class to be unloaded as a result of a low- or no-memory condition. In the preferred embodiment, the order in which currently loaded classes are unloaded by the memory management handler 27 is set using the setFreeAppsFirst( ) method. The setFreeAppsFirst( ) method allows a client to set or change the order in which applications are unloaded in case of a low- or no-memor condition. This may be accomplished in any of several ways, including by adding an unload priority field to each entry in the application list which indicates the ranking of the application for unloading the class. By caching the class, the Application Manager 24 maintains a reference to the class, thereby forcing the Java Virtual Machine 22 not to garbage collect the class 28a, 28b. The application class 28a, 28b can then be re used to create new instances 30a, 30b, 30c of the class 28a, 28b, and re-execute application functionality.
[0046] Application Manager 24 also preferably allows caching of application data in a data cache 54 in memory 50, even after an application has terminated and/or been unloaded. An example is Java applications in the electronic test domain. The first execution of the application could save setup informatio which allow future running of the application to execute faster. In this example, Application Manager 24 could cache test system calibration information Again, by caching the test system calibration information in data cache 54, Application Manager 54 maintains a reference to the data, thereby forcing the Java Virtual Machine not to garbage collect the data. Accordingly, data caching in this manner allows information to be saved for subsequent runs of the same or different application.
[0047] Application Manager 24 attempts to have applications 26a, 26b, 26c continue to execute in low- or no-memory situations. If, during the execution of an application 26a, 26b, 26c, the JVM 22 runs out of memory, an OutOfMemoryError error is generated. Application manager 24 includes a memory management handler 27 which handles low- or no-memory conditions. In the preferred embodiment, memory management handler 27 is triggered into action by the occurrence of an OutOfMemoryErorr generated by the JVM 22. In one embodiment, memory management handler 27 reacts by judiciously dumping class objects 28a, 28b and application objects 30a, 30b and 30c from its application cache. The order for unloading cached class object 28a, 28b and application objects 30a, 30b and 30c is set using the setFreeAppsFirst( ) method. Alternatively, the order is determined according to a predetermined algorithm coded within the memory management handler 27 itself. By calling the Java(R) Runtime.gc( ) method, Application Manager informs the JVM 22 that the unloaded class objects 28a, 28b and application objects 30a, 30b and 30c are available to be reclaimed. The unloading of class objects 28a, 28b from the application cache 52 forces the application objects 30a, 30b, 30c associated with the class 28a, 28b to be terminated and unloaded as well. Accordingly, if startup speed of application execution is important, it may be preferable to unload application objects 30a, 30b and 30c before unloading class objects 28a, 28b. On the other hand, if it is important to keep a particular application running regardless, if may be preferable to unload all application objects of another class along with the class associated with those objects in order to free up enough memory to keep the particular application running. Solutions to having applications terminate is to separately monitor memory levels reacting to low memory conditions, or to checkpoint application results allowing applications to be restarted continuing after a checkpoint. These approaches may allow executing applications to continue uninterrupted, or at least as close to uninterrupted as possible.
[0048] Application Manager 24 preferably provides memory management functionality. In one embodiment, Application Manager 24 references class objects 28a, 28b, application objects 30a, 30b, 30c and global data in respective application 52 and data 54 caches, thereby ensuring that their associated objects are not garbage collected.
[0049] In an alternative embodiment, memory management handler 27 continuously monitors the free memory level, and in times of low- or no-free memory, removes objects from its caches 52, 54 and triggers garbage collection to free up more memory as needed. As previously described, garbage collection is triggered by calling the Java(R) Runtime.gc( ) method after removing items from the application cache 52 o data cache 54. A choice must be made when implementing an Application Manager 24 as to how it manages the memory 50 in low memory situations. One choice is which cache, application cache 52 or data cache 54, is to have items removed from it first when low free memory occurs. In most cases it may be easier to remove application objects 30a, 30b, 30c and associated class objects 28a, 28b, and re-downloac them as needed. This decision is predicated on the long amount of time it may take to recreate global data.
[0050] Application and global data objects are preferably designed to be as separate as possible for the caching scheme to work correctly. If a global data class is referenced as a member of a class object 28a, 28b, then unreferencing the data class will do nothing to free up memory unless the class object 28a, 28b is also unreferenced. The reverse case of a global data class referencing a class object 28a, 28b also causes the same result. Class objects 28a, 28b should only reference global data classes in methods, not in class members. Preferably, global data objects never reference class objects 28a, 28b.
[0051] FIG. 4 is an operational flowchart illustrating an exemplary embodiment of the management of application class cache 52. Application Manager 24 receives 402 a request from a client 12. If the executio of the request would result in the use of free memory (e.g., loadAppl( ), loadAndinit( ), Initappl( )), a determination 404 is made as to whether the execution of the received request would result in a low- or no-memory condition. If so, class object(s) 28a, 28b and/or application object(s) 30a, 30b, 30c are selected

406 to unload from the application cache 52. The selected object(s) are then unloaded 408 from the application cache 52. If the received request is a loadAppl( ) or loadAndinitAppl( ) request, a class object is loaded from the network 14 and defined 410. If the request is a loadApp( ) request, the request is then complete. If the request is a loadAndinitAppl( ) request or an InitAppl( ) request, an application object 30a, 30b, 30c is instantiated 412 and the request is complete. If the request is a startAppl( ) request, the application is started 414, typically by calling a main( ) method on the application object 30a, 30b, 30c. If the request is a stopAppl( ) request, the application is stopped 416. If the request is an unload( ) request, the class object 28a, 28b is unloaded 418. If the request is a setFreeMemoryLimit( ) request, the free memory threshold, upon which a low-memory condition is based, is set 420. If the request is a setFreeAppsFirst( ) request, the order in which application classes28a, 28b and object classes 30a, 30b, 30c are selected for unloaded upon the detection of a low- or no-memory condition is set 422. If the reques is a query (e.g., which application classes are loaded (applClasses( )), which applications are initialized (applications( )), which applications are currently running (applInstances( )), how much available memory exists (freeMemory( )), how much total memory exists (totalMemory( )), what the existing free memory limit threshold is (getFreeMemoryLimit( )), or what the existing order for unloading class and application objects is (getFreeAppsFirstPolicy( ))), the query is executed 424 with the requested parameter returned.

[0052] When it is determined 404 that a low- or no-memory situation exists, in the illustrative embodiment Application Manager 24 removes references to the objects that are chosen to be unloaded and causes the unreferenced objects to be removed from the application cache 52 by calling the Java(R) Runtime.gc( ) method. As described earlier, only unreferenced classes 28a, 28b are removed from the cache 52 because only unreferenced classes can be garbage collected by the Java(R) Runtime.gc( ) method. In some JVM 2: implementations, the class's ClassLoader object may also have to be unreferenced for the class to be garbage collected.

[0053] FIG. 5 is a class diagram illustrating an example implementation of a system 500 which employs the invention. A description of each of the classes defined and used in the example implementation of FIG. 5 is shown below in Table 1. Table 2 contains a description of the main attributes defined in each of the classes used in the example implementation of FIG. 5. As illustrated in FIG. 5, class ApplMgr 502 manages one-to-many instances of Application class 510. Class ApplMgr 502 uses a ServerProtocol class 504 which talks to a ClientProtocol class 506 in order to interface with a client class ApplClient 508. As described in Table 1, class ApplMgr 502 is the class that implements the Application Manager 24. Application class 510 models a single application. ServerProtocol class 504 and ClientProtocol class 506 implements client interface 4. Together they encapsulate the network protocol 48 between the Application Manager class ApplMgr 502 and client class ApplClient 508.

<tb><sep>TABLE 1
<tb><sep>Class<sep>Description
<tb><sep>ApplMgr<sep>Overall Application Manager class
<tb><sep>Application<sep>Models a single application
<tb><sep>ServerProtocol,<sep>Encapsulates the network protocol between ApplMgr and
<tb><sep>ClientProtocol<sep>ApplClient classes representing the interface supported
<tb><sep><sep>by the ApplMgr.
<tb><sep>ApplBase<sep>Interface defining methods which an Application must
<tb><sep><sep>define to be used by the ApplMgr
<tb><sep>ApplClient<sep>Example client side application to interact with an
<tb><sep><sep>ApplMgr class
<tb><sep>TABLE 1
<tb><sep>Class<sep>Description
<tb><sep>ApplMgr<sep>Overall Application Manager class
<tb><sep>Application<sep>Models a single application
<tb><sep>ServerProtocol,<sep>Encapsulates the network protocol between ApplMgr and
<tb><sep>ClientProtocol<sep>ApplClient classes representing the interface supported
<tb><sep><sep>by the ApplMgr.
<tb><sep>ApplBase<sep>Interface defining methods which an Application must
<tb><sep><sep>define to be used by the ApplMgr
<tb><sep>ApplClient<sep>Example client side application to interact with an
<tb><sep><sep>ApplMgr class

[0054] FIG. 6 is a block diagram of system 500 illustrating the communications between the classes to perform the loading of a new class. Table 3 specifies the initial assumptions, the results of the load operation, and the required agents for performing the load. As described, it is assumed that an application manager class ApplMgr 502 is running in the JVM 22 of the embedded device 20 and that a user has requested a download by instructing the ApplClient 508 running on the client computer 12 to download the new class to the embedded device 20.

<tb><sep>TABLE 3
<tb><sep>Assumptions<sep>ApplMgr is already running in the JVM.
<tb><sep>Results<sep>The classes associated with an application are loaded into
<tb><sep><sep>the JVM, an instance of the "main class" is created, and the

<tb><sep><sep>init() method is called on this new instance.
<tb><sep>Agents<sep>User = Instructs ApplClient to download new application
<tb><sep><sep>class to embedded device running ApplMgr
<tb><sep><sep>ApplClient = Host side program downloading appl classes
<tb><sep><sep>ClientProtocol = Host side class handling network com-
<tb><sep><sep>munication for client
<tb><sep><sep>ServerProtocol = Server side class handling network com-
<tb><sep><sep>munication for ApplMgr
<tb><sep><sep>ApplMgr = Application Manager class
<tb><sep><sep>Application = Application class modeling a downloaded
<tb><sep><sep>application

[0055] As illustrated in FIG. 6, a download operation begins with a user request 602 from a client user application such as client user class ApplClientU 516 to client application ApplClient 508. In this example, client user class ApplClientU 516 generates a download request by conforming to the protocol of transmitting a 'D' character followed by the name of the file containing the class data followed by the network node to which the class data is to be transmitted. In this example, the class data is named 'test' and the network node address of the embedded device 20 is named 'HP'. The request 602 causes the ApplClient 508 to be run to download class 'test' to node 'HP'. ApplClient 508 reads the class data contained in 'test' into memory, then calls 604 ClientProtocol.loadClass( ) method to download the class. ClientProtocol 506 forms a network 14 connection to ServerProtocol on node 'HP', and sends 606 the class data associated with class data file 'test' over network 14. ServerProtocol 504 decodes the request and class data, then calls 608 ApplMgr.loadClass( ) method to load the class. ApplMgr 502 creates 610 an Application object for the class if needed (i.e., the class is not already cached in the application cache). ApplMgr 502 then calls 612 the loadClass( ) method on Application 510, passing it all class data associated with class 'test'. The new Application object is added 616 to the ApplMgr cache of Applications and application list ApplMgr.iapps 23 is updated 616 to reflect the addition of class 'test' to its application objects that it manages. If the class 'test' being loaded is a main class, Application defines and resolves the class, creates an instance of the class, and calls the instance's init( ) method, 618. A true result is passed 620 back to the ApplClient signaling successful loading of the class.

[0056] In implementation, the ServerProtocol class 504 may, if desired, create a separate thread for handling each request, allowing multiple requests to be processed concurrently. If implemented in this manner, the methods implemented by classes ApplMgr 502 and Application 510 must be synchronized appropriately.

[0057] If an Application 510 is made up of multiple classes, a loadClass( ) request will be made for each class. This requires the ApplMgr 502 to be able to locate and add classes to an existing Application object 510. Applications are identified by a unique name, identical to the name of the main class. By convention, base classes and interfaces are loaded prior to loading the main class to ensure that the instance of the main class is created correctly. Another approach would be to create a new ApplMgr method to create an instance of an application. This would allow multiple applications of the same class to be executing concurrently.

[0058] FIG. 7 is a block diagram of system 500 illustrating the communications between the classes to perform the starting of an application. Table 4 specifies the initial assumptions, the results of the start operation, and the required agents for performing the start. As described, it is assumed that an application manager class ApplMgr is running in the JVM 22 of the embedded device 20 and that a user has requested a start by instructing the ApplClient running on the user's machine to start the application on embedded device 20.

<tb><sep>TABLE 4
<tb><sep>Assumes<sep>ApplMgr is already running in the JVM. The application
<tb><sep><sep>classes have been previously loaded by the ApplMgr.
<tb><sep>Results<sep>The desired Application starts executing.
<tb><sep>Agents<sep>User = Instructs ApplClient to start application 'test' on node
<tb><sep><sep>'HP'.
<tb><sep><sep>ApplClient = Host side program starting Applications
<tb><sep><sep>ClientProtocol = Host side class handling network communi-
<tb><sep><sep>cation for client
<tb><sep><sep>ServerProtocol = Server side class handling ApplMgr network
<tb><sep><sep>communication
<tb><sep><sep>ApplMgr = Application Manager class
<tb><sep><sep>Application = Application class modeling the application to be
<tb><sep><sep>started
<tb><sep><sep>ApplBase = Application instance being started

[0059] As illustrated in FIG. 7, a start operation begins with a user request 702 from a client user application such as client user class ApplClientU 516 to client application ApplClient 508. In this example, client user class ApplClientU 516 generates a start request by conforming to the protocol of transmitting a 'B' character (for 'begin') followed by the application name 'test' followed by the network node 'HP' to which the class

data is to be transmitted. The request 702 causes the AppiClient 508 to be run to start application 'test' on node 'HP'. ApplClient 508 calls 704 ClientProtocol.startAppl( ) method to start application. ClientProtocol 506 forms a network connection to ServerProtocol 504 on node 'HP', and sends 706 a start message. ServerProtocol 504 decodes the start request, then calls 708 ApplMgr.startAppl( ) method. ApplMgr 502 locates the correct Application object 510, then calls 710 the startAppl( ) method on Application 510. The Application object 510 creates 712 a new Thread 518 to execute 714 its own run( ) method. Within the run ( ) method the ApplBase instance's 514 main method is called 716, passing any parameters. If no parameters are to be passed to the Application instance, the run( ) method could be called directly. Application object 510 then changes 718 the state of the application instance to executing.

[0060] In an alternative embodiment, an ApplClass object (i.e., a class loader 512) manages Application classes which model an instance of an Application class 510.

[0061] FIG. 8 is a block diagram of system 500 illustrating the communications between the classes to perform the stopping of an application. Table 5 specifies the initial assumptions, the results of the stop operation, and the required agents for performing the stop. As described, it is assumed that an application manager class ApplMgr is running in the JVM 22 of the embedded device 20 and that a user has requestec a stop by instructing the ApplClient running on the user's machine to stop the application on embedded device 20.

<tb><sep>TABLE 5
<tb><sep>Assumes<sep>ApplMgr is already running in the JVM.
<tb><sep><sep>The application classes have been previously loaded by the
<tb><sep><sep>ApplMgr
<tb><sep><sep>The application is already running.
<tb><sep>Results<sep>The desired Application starts executing.
<tb><sep>Agents<sep>User = Instructs ApplClient to stop application test on node
<tb><sep><sep>HP.
<tb><sep><sep>ApplClient = Host side program starting Applications
<tb><sep><sep>ClientProtocol = Host side class handling network communi-
<tb><sep><sep>cation for client
<tb><sep><sep>ServerProtocol = Server side class handling ApplMgr network
<tb><sep><sep>communication
<tb><sep><sep>ApplMgr = Application Manager class
<tb><sep><sep>Application = Application class containing ApplBase instance
<tb><sep><sep>to be stopped
<tb><sep><sep>ApplBase = the executing application instance which is to be
<tb><sep><sep>stopped

[0062] As illustrated in FIG. 8, a stop operation begins with a user request 802 from a client user applicatior such as client user class ApplClientU 516 to client application ApplClient 508. In this example, client user class ApplClientU 516 generates a stop request by conforming to the protocol of transmitting an 'E ' character (for 'end') followed by the application name 'test' followed by the network node 'HP' to which the class data is to be transmitted. The request 802 causes the ApplClient 508 to be run to stop execution of application 'test' on node 'HP'. ApplClient 508 calls 804 ClientProtocol.stopAppl( ) method to stop the application. ClientProtocol 506 forms a network connection to ServerProtocol 504 on node 'HP', and sends 806 a stop message. ServerProtocol 504 decodes the stop request, then calls 808 ApplMgr.stopAppl( ) method. ApplMgr 502 locates the correct Application object 510, then calls 810 the stopAppl( ) method on Application 510. The Application object 510 locates the ApplBase 514 instance by the name passed into stopAppl( ) and calls 812 its terminate( ) method. Application then uses Thread.join( ) to ensure the ApplBase has terminated. Application object 510 changes 814 the state of the ApplBase 514 instance to terminated.

[0063] Stopping an ApplBase 514 instance is an asynchronous event because the application is executing in its own thread. It may be desirable to not actually wait for the application to stop, because if the application does not poll its terminate flag, it will not stop. Accordingly, in implementation, the design may alternatively use a join( ) method with a timeout and then force termination by calling stop( ).

[0064] FIG. 9 is a block diagram of system 500 illustrating the communications between the classes when a low- or no-memory condition occurs during the execution of an instance of an Application 510. Table 6 specifies the initial assumptions, the results of an out-of-memory condition, and the required agents for performing handling the condition. As described, it is assumed that an application manager class ApplMgr 502 is running in the JVM 22 of the embedded device 20 and that one or more application classes are loaded and executing.

<tb><sep>TABLE 6
<tb><sep>Assumptions<sep>ApplMgr is already running in the JVM.
<tb><sep><sep>One or more application classes are loaded and executing.
<tb><sep>Results<sep>The JVM and ApplMgr continue to execute, and free up
<tb><sep><sep>memory by managing user application class and instance
<tb><sep><sep>caches.
<tb><sep>Agents<sep>User = Instructs ApplClient to stop application 'test' one<sep>e

<tb><sep><sep>node 'HP'.
<tb><sep><sep>ApplClient = Host side program starting Applications
<tb><sep><sep>ClientProtocol = Host side class handling network commu-
<tb><sep><sep>nication for client
<tb><sep><sep>ServerProtocol = Server side class handling ApplMgr net-
<tb><sep><sep>work communication
<tb><sep><sep>ApplMgr = Application Manager class
<tb><sep><sep>Application = Application class containing ApplBase
<tb><sep><sep>instance to be stopped
<tb><sep><sep>ApplBase = The executing application instance which is to
<tb><sep><sep>be stopped

[0065] As illustrated in FIG. 9, an instance ApplBase 514 of a running Application 510 generates 902 an OutOfMemoryError. Preferably, the OutOfMemoryError is handled 904 by memory management handler 27 within the Application.run( ) method. The handler 27 notifies the ApplMgr 502 of the condition by calling 906 an ApplMgr.outOfMemory( ) method. Application 510 then returns from run( ) to stop 908 the ApplBase 514 thread. ApplMgr 502 iterates through its Application cache 52, finding classes which do not have executing ApplBases by calling 908 Application.applCount( ) method. Those Applications without executing ApplBases are instructed 910 to freeMemory( ), which causes them to dump their class from application cache 52, and set all object references to null. ApplMgr 502 then removes 912 those Applications from its application cache 52, and calls 914 Runtime.gc( ) to cause garbage collection to collect the unreferenced objects, thereby freeing up memory.

[0066] As described in detail above, the present invention provides an application manager and API specification which runs within a Java Virtual Machine that is intended for use in electronics devices or appliances and in other embedded systems having environments with limited resource constraints. The invention provides the flexibility to cache and terminate applications for future runs when memory resources become constrained in order to free up memory for higher-priority applications.

Management program of JAVA application program for embedded system

Claims of corresponding document: **US6430570** Translate this text

What is claimed is:

[0067] 1. An application manager which manages one or more applications in an embedded device, said embedded device comprising an application cache for storing class objects, an interface for communicating with a remote client, and a hardware independent processor which processes code written in a hardware independent language, said hardware independent processor installed and running on said embedded device and capable of downloading and executing code written in said hardware independent language on said embedded device, said application manager comprising:a class object list having one or more class object entries each of which identifies a class object currently loaded in said embedded device; an instance list having one or more instance entries each of which identifies an instance of a corresponding one of said class objects currently loaded in said embedded device; a class loader method which loads an application class received from said client via said interface, creates a new class object for said application class, stores said new class object in said application cache, adds a class object entry to said class object list identifying said new class object as being loaded in said application cache, instantiates one or more instances of said new class object, and adds an instance entry to said instance list identifying each of said instantiated one or more instances of said new class object.

[0068] 2. An application manager in accordance with claim 1, wherein:each class object identified in said class object list maintains its own instance list.

[0069] 3. An application manager in accordance with claim 1, wherein:each instance entry in said instance list comprises an execution state attribute indicating a current execution state of its corresponding instance.

[0070] 4. An application manager in accordance with claim 3, comprising:a start application method which creates a new instance of a class object and adds said new instance entry to said instance list if said instance to be started does not exist, causes said instance to be started to begin executing, and updates said execution state attribute corresponding to said instance entry in said instance list of said instance to be started to indicate that said instance to be started is currently executing.

[0071] 5. An application manager in accordance with claim 3, comprising a stop application method which causes an instance to be stopped which is currently executing on said embedded device to stop executing, and updates said execution state attribute corresponding to said instance entry in said instance list of said instance to be stopped to indicate that said instance to be stopped is not executing.

[0072] 6. An application manager in accordance with claim 1, comprising:a memory management handler responsive to the detection of a low- or out-of-memory condition which selects a class object cached in said application cache to be unloaded, and unloads said selected class object from said application cache.

[0073] 7. An application manager in accordance with claim 3, comprising:a query application method responsive to an indication of an instance of a class object to be queried which indicates said execution state attribute corresponding to said instance to be queried.

[0074] 8. An application manager in accordance with claim 1, comprising:a start application method which creates a new instance of a class object and adds said new instance to said instance list if said instance to be started does not exist, and causes said instance to be started to begin executing.

[0075] 9. An application manager in accordance with claim 1, comprising a stop application method which causes an instance to be stopped which is currently executing on said embedded device to stop executing.

[0076] 10. An application manager in accordance with claim 1, comprising an unload application method which causes an instance to be unloaded to be removed from said application cache and said instance entry corresponding to said instance to be unloaded to be removed from said instance list.

[0077] 11. A method for managing applications in an embedded device, said method comprising:receiving an application class; creating a new class object for said application class; storing said new class object in an application cache in said embedded device; adding a class object entry to a class object list maintained in said embedded device identifying said new class object as being loaded in said application cache;

instantiating one or more instances of said new class object; and for each instantiated one or more instances of said new class object, adding an instance entry to an instance list maintained in said embedded device identifying each of said instantiated one or more instances of said new class object.

[0078] 12. A method in accordance with claim 11, comprising:maintaining a separate instance list for each class object identified in said class object list.

[0079] 13. A method in accordance with claim 11, comprising:receiving a start application request indicating an instance of a class object to be started; if said instance to be started does not exist, instantiating said instance of said class object to be started and adding an instance entry corresponding to said instance to be started to said instance list; and causing said instance to be started to begin executing.

[0080] 14. A method in accordance with claim 11, comprising:receiving a stop application request indicating an instance of a class object to be stopped; and if said instance to be stopped exists and is currently executing, causing said instance to be stopped to stop executing.

[0081] 15. A method in accordance with claim 11, comprising:receiving an unload application request indicating an instance of a class object to be unloaded; and if said instance to be unloaded exists, causing said instance to be unloaded to be removed from said application cache and said instance entry corresponding to said instance to be unloaded to be removed from said instance list.

[0082] 16. A method in accordance with claim 11, comprising:for each instance entry in said instance list, maintaining an execution state attribute indicating a current execution state of said instance corresponding to said instance entry.

[0083] 17. A method in accordance with claim 16, comprising:receiving a start application request indicating an instance of a class object to be started; if said instance to be started does not exist, instantiating said instance of said class object to be started and adding an instance entry corresponding to said instance to be started to said instance list; causing said instance to be started to begin executing; and updating said execution state attribute of said instance entry in said instance list of said instance to be started to indicate that said instance to be started is currently executing.

[0084] 18. A method in accordance with claim 16, comprising:receiving a stop application request indicating an instance of a class object to be stopped; if said instance to be stopped exists and is currently executing, causing said instance to be stopped to stop executing; and updating said execution state attribute corresponding to said instance entry in said instance list of said instance to be stopped to indicate that said instance to be stopped is not executing.

[0085] 19. An application manager in accordance with claim 16, comprising:receiving a query application request indicating an instance of a class object to be queried; if said instance to be queried exists, indicating said execution state attribute corresponding to said instance to be queried.

[0086] 20. A method in accordance with claim 11, comprising:detecting a low- or out-of-memory condition; selecting a class object loaded in said application cache to be unloaded; and unloading said selected class object from said application cache.

---

[19] 中华人民共和国国家知识产权局

[51] Int. Cl[7]
G06F 9/48

[12] 发明专利说明书

[21] ZL 专利号 00103710.2

[45] 授权公告日 2005 年 4 月 13 日

[11] 授权公告号 CN 1197004C

[22] 申请日 2000.3.1 [21] 申请号 00103710.2

[30] 优先权

[32] 1999. 3. 1 [33] US [31] 09/259616

[71] 专利权人 惠普公司

地址 美国加利福尼亚州

[72] 发明人 F·P·贾奇 C·-C·多兰

审查员 盖 浩

[74] 专利代理机构 中国专利代理(香港)有限公司

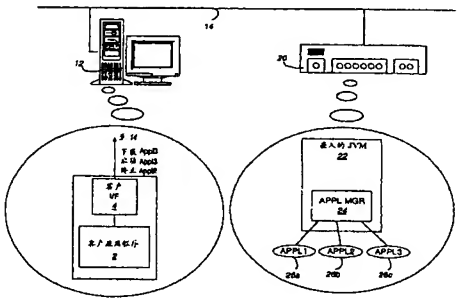
代理人 王 勇 王忠忠

权利要求书 2 页 说明书 22 页 附图 9 页

[54] 发明名称 用于管理嵌入设备中的应用程序的方法

[57] 摘要

一个 Java 应用程序管理程序(24)控制在一个单个 Java 虚拟机(JVM)(22)内运行的 Java 应用程序(26a、26b、26c)的下载、执行和高速缓存器。应用程序管理程序(24)接受应用程序类下载(load( )/Loadandlnit( ))、实例应用程序类(lnit( ))和控制应用程序(26a、26b、26c)的生存周期。



ISSN1008-4274

知识产权出版社出版

自一个客户的每个请求；以及当 Serlet 被停止（即 Web 服务器关闭）时调用 destroy()。Servlet API 不直接支持存储器管理或提供一个管理 Servlets 的公共包。

HP 公司的 Embedded Java Lab SmallWeb 提供一个在 JVM 中执行的至基于 Java 目标的基于 Web 的接口。SmallWeb 当需要时可以装载目标，同时对目标提供一个把它们的功能（通过方法调用）转出到 Web 浏览器的装置。SmallWeb 不直接提供停止应用程序目标或存储器管理特性。此外，SmallWeb 通常要求一个文件系统，同时对某些嵌入环境，SmallWeb 的辅助操作要求可能太大。

在许多嵌入领域中使用 Java®的一个问题是 Java®存储器子系统的非确定的方面。Java®语言的非确定存储器管理方案允许通过一个垃圾回收站方法 gc()回收未引用目标；然而，垃圾回收站没有规定如何或何时回收这些目标。原来的应用程序一般通过调用 Java® Runtime 类 gc()方法来使垃圾回收站出现。如果 JVM 运行存储器溢出，则出现一个 OutofMemory-Error 错误。虽然不必管理存储器是 Java 语言的一个优点，但大多数嵌入应用程序需要在如何管理存储器方面较之在 Java®语言中目前提供的更严格的控制。需要更严格的控制是因为有些嵌入应用程序必须继续在低存储器环境中执行。这一般通过在原来的非 Java 嵌入应用程序中补充一个存储器管理程序来解决，以确保为了继续执行有足够存储器仍是可用的。

因此，为了下载到 Java 使能嵌入设备中和控制在 Java 使能嵌入设备中的应用程序的寿命，对通用方法存在一个需求。对一个存储器管理处理程序也存在一个要求，存储器管理处理程序处理在设备上执行应用程序期间检测到的低存储器或没有存储器的情况，同时该处理程序按照基于优先的算法释放存储器。

本发明是一个新颖的系统和方法，用于管理在存储器受限制的嵌入设备环境中应用程序的下载和生存周期。一个基于 Java 的应用程序管理程序控制在嵌入设备上安装的单个 Java 虚拟机（JVM - Java Virtual Machine）内运行的 Java 应用程序的下载、执行和高速缓存器。一个有网络能力的应用程序接口（API - Application Program Interface）被规定，它提供装载类文件、启动、初始化和停止应用程序执行的功能，以及在 Java®使能嵌入设备中存储器管理的功能。

服务器以便服务该请求．客户应用程序 2 可以在计算机系统 12 上它自己的环境中执行，或者用另一个办法在一个 Java 使能 Web 浏览器（未示出）内执行，该浏览器包含它自己的 Java 虚拟机（JVM－Java Virtual Machine）（未示出）．Web 浏览器用嵌入的 Java 小应用程序解释 Web 文件，Java 小应用程序规定主客户应用程序小应用程序类文件的地点．Web 浏览器启动它的 JVM 并传送客户应用程序小应用程序类文件的地点到它自己的类装入程序 6．每个类文件知道它请求的任何附加的类文件的名字．这些附加的类文件可以来自于网络 14（即与网络 14 相连的其它机器）或客户计算机系统 12．

嵌入设备 20 是一个 Java 使能设备，具有安装在它上面的 Java 虚拟机（JVM）22．图 2 是更详细说明嵌入设备的一个方块图．嵌入设备 20 包括一个 JVM22、一个由应用程序高速缓存器 52 和数据高速缓存器 54 组成的存储器 50 以及一个网络接口 25．JVM22 包括一个类装入程序 42 和一个执行单元 46．最好 JVM22 也包括一个字节代码检验器 44，虽然在有些存储器受限制的设备中这不可能实现．按照本发明，类装入程序 42 采用本发明的应用程序管理程序 24 来实现．应用程序管理程序 24 是一个负责下载、执行和高速缓存器其它基于 Java 的程序的 Java 程序．当 JVM22 开始在嵌入设备 20 上执行时，它开始执行应用程序管理程序 24．

应用程序管理程序 24 是一个服务器设备，用作一个基于网络的类装入程序，能通过网络 14 经由一个网络协议 48 接收应用程序类文件 40，并把在应用程序类文件 40 中包含的应用程序类传到文件中．如本专业技术人员所知，一个类文件 40 是包括与硬件体系结构无关的由 Java®编译程序产生的字节代码的一个文件．字节代码仅由 JVM 例如 JVM22 才可执行．应用程序管理程序 24 是一个非典型的基于网络的 Classloader，根据需要从网络 14 装载类 40．不是象由典型类装入程序通常所做的那样通过 Java® load Class()方法装载类文件 40，应用程序管理程序 24 代之以在服务器套接字或其它协议 48 上等待下载类文件 40 的一个请求．当接收到一个下载请求时，应用程序管理程序 24 从网络 14 接收类文件 40，把它传到类 28a、28b，然后调用 Java® ClassLoader.define class()方法．类 28a、28b 应被解析以确保通过调用 Java® Class Loader、resolve Class()方法

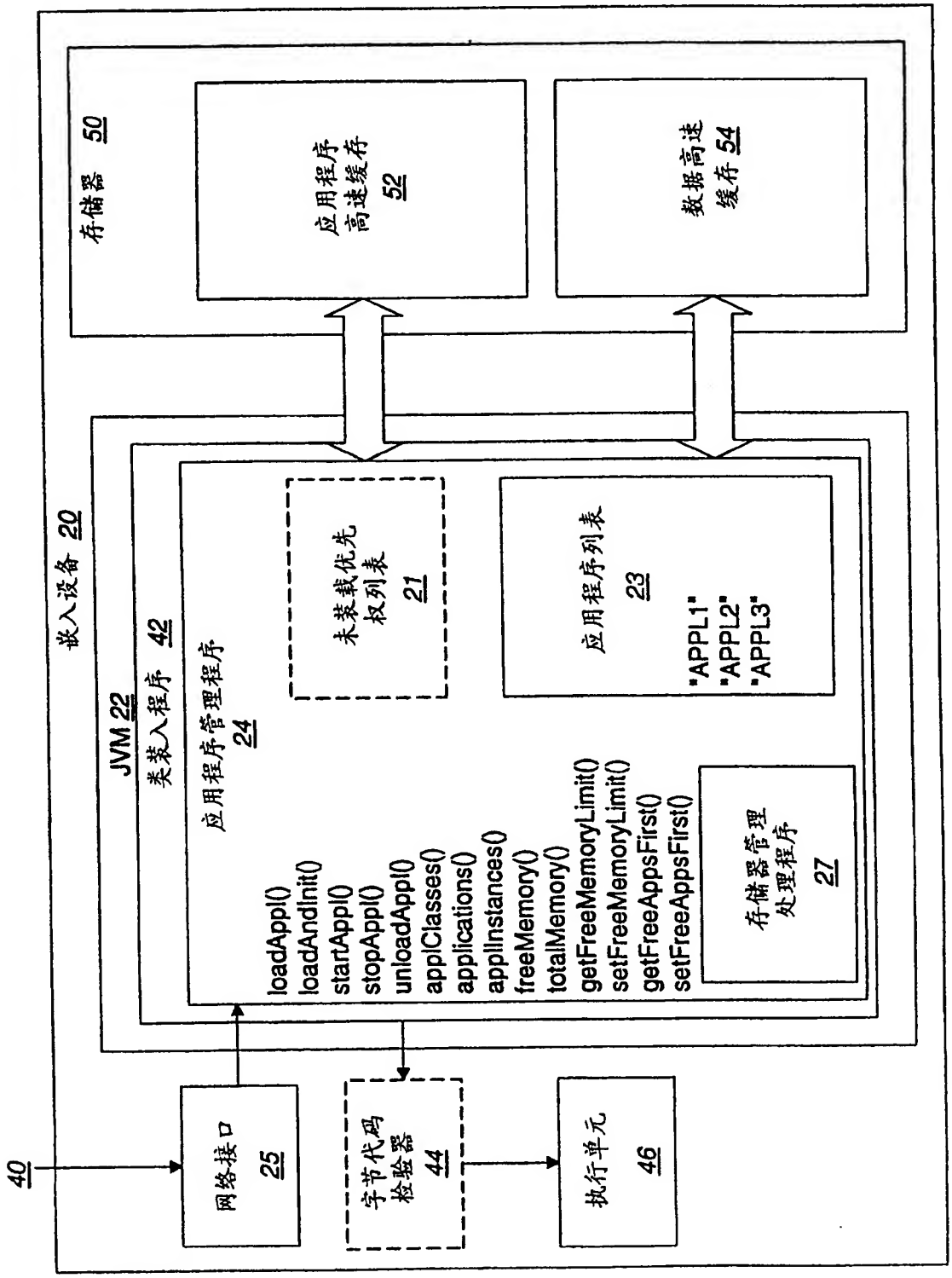

40
嵌入设备 20
网络接口 25
字节代码检验器 44
执行单元 46
JVM 22
类装入程序 42
应用程序管理程序 24
loadAppl()
loadAndInit()
startAppl()
stopAppl()
unloadAppl()
applClasses()
applications()
applInstances()
freeMemory()
totalMemory()
getFreeMemoryLimit()
setFreeMemoryLimit()
getFreeAppsFirst()
setFreeAppsFirst()
未装载优先权列表 21
应用程序列表 23
"APPL1"
"APPL2"
"APPL3"
存储器管理处理程序 27
存储器 50
应用程序高速缓存 52
数据高速缓存 54

图 2